Outlook

- 1: syntax + transformation
- 2: extraction + analysis
- 3: code generation
Syntax & transformation
Syntax

S
NP  VP
Pronoun  V  NP
I  know  that
S
NP  VP
NP  V  NP
Jane  thinks  that
S
NP  VP
NP  V  NP
Bill  likes  N
   beer
Syntax definition

```plaintext
module Syntax
extend lang::std::Layout;

start syntax Controller =
  controller:
    Events events
    ResetEvents? resets
    Commands? commands
    State+ states;

syntax Events
  = @Foldable "events" Event* "end";
syntax ResetEvents
  = @Foldable "resetEvents" Id* "end";
syntax Commands
  = @Foldable "commands" Command* "end";
```
module Syntax
extend lang::std::Layout;

start syntax Controller = controller:
  Events events
  ResetEvents? resets
  Commands? commands
  State+ states;

syntax Events = @Foldable "events" Event* "end";
syntax ResetEvents = @Foldable "resetEvents" Id* "end";
syntax Commands = @Foldable "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
    controller:
        Events events
        ResetEvents? resets
        Commands? commands
        State+ states;

syntax Events
    = @Foldable "events" Event* "end";
syntax ResetEvents
    = @Foldable "resetEvents" Id* "end";
syntax Commands
    = @Foldable "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
controller:
  Events events
  ResetEvents? resets
  Commands? commands
  State+ states;

syntax Events
  = @Foldable "events" Event* "end";
syntax ResetEvents
  = @Foldable "resetEvents" Id* "end";
syntax Commands
  = @Foldable "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
controller:
Events events
ResetEvents? resets
Commands? commands
State+ states;

syntax Events
  = @Foldable "events" Event* "end";
syntax ResetEvents
  = @Foldable "resetEvents" Id* "end";
syntax Commands
  = @Foldable "commands" Command* "end";
Syntax definition

module Syntax
extend lang::std::Layout;

start syntax Controller =
controller:
  Events events
  ResetEvents? resets
  Commands? commands
  State+ states;

syntax Events
  = @Foldable "events" Event* "end";
syntax ResetEvents
  = @Foldable "resetEvents" Id* "end";
syntax Commands
  = @Foldable "commands" Command* "end";
Lexical syntax

\[\text{lexical Id} = ([a-zA-Z][a-zA-Z0-9_]\star !>> [a-zA-Z0-9_])\]
\Reserved ;

\[\text{keyword Reserved} = \"events\" \mid \"end\" \mid \"resetEvents\" \mid \"state\" \mid \"actions\" ;\]
Lexical syntax

lexical Id
    = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\ Reserved ;

keyword Reserved
    = "events"
    | "end"
    | "resetEvents"
    | "state"
    | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]*) !>> [a-zA-Z0-9_])
  \ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Lexical syntax

lexical Id
= ([a-zA-Z][a-zA-Z0-9_]* ![>>] [a-zA-Z0-9_])
\ Withdrawed ;

keyword Reserved
= "events"
| "end"
| "resetEvents"
| "state"
| "actions" ;

lexicals don’t get layout

follow restriction

character class

keyword reservation
Lexical syntax

lexical Id
= ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_]) \ Reserved ;

keyword Reserved
= "events"
| "end"
| "resetEvents"
| "state"
| "actions" ;
Parse trees
Abstract Syntax

data Controller = controller(list[Event] events,
                           list[str] resets,
                           list[Command] commands,
                           list[State] states);

data State = state(str name,
                   list[str] actions,
                   list[Transition] transitions);

data Command = command(str name, str token);
data Event = event(str name, str token);
data Transition = transition(str event, str state);
Abstract Syntax

```plaintext
data Controller = controller(list[Event] events,
                           list[str] resets,
                           list[Command] commands,
                           list[State] states);

data State = state(str name,
                   list[str] actions,
                   list[Transition] transitions);

data Command   = command(str name, str token);
data Event      = event(str name, str token);
data Transition = transition(str event, str state);
```

non-terminals map to data type
Abstract Syntax

```
data Controller
  = controller([Event] events,
                [str] resets,
                [Command] commands,
                [State] states);

data State
  = state(str name,
           [str] actions,
           [Transition] transitions);

data Command
  = command(str name, str token);
data Event
  = event(str name, str token);
data Transition
  = transition(str event, str state);
```
Abstract Syntax

non-terminals map to data type

```plaintext
data Controller = controller(list[Event] events,
                         list[str] resets,
                         list[Command] commands,
                         list[State] states);

data State = state(str name,
                    list[str] actions,
                    list[Transition] transitions);

data Command = command(str name, str token);
data Event = event(str name, str token);
data Transition = transition(str event, str state);
```

regulars (+/*/?) map to lists

Wednesday, May 23, 12
Abstract Syntax

non-terminals map to data type

data Controller = controller(list[Event] events,
  list[str] resets,
  list[Command] commands,
  list[State] states);

data State = state(str name,
  list[str] actions,
  list[Transition] transitions);

data Command = command(str name, str token);

data Event = event(str name, str token);

data Transition = transition(str event, str state);

regulars (+/*/?) map to lists

lexicals to str/ int/real/bool

controller(
    [  
        event("doorClosed","D1CL"),  
        event("drawerOpened","D2OP"),  
        event("lightOn","L1ON"),  
        event("doorOpened","D1OP"),  
        event("panelClosed","PNCL")  
    ],  
    ["doorOpened"],  
    [  
        command("unlockPanel","PNUL"),  
        command("lockPanel","PNLK"),  
        command("lockDoor","D1LK"),  
        command("unlockDoor","D1UL")  
    ],  
    state(  
        "idle",  
        ["unlockDoor","lockPanel"],  
        [transition("doorClosed","active")]),  
    state(  
        "active",  
        [],  
        [  
            transition("drawerOpened","waitingForLight"),  
            transition("lightOn","waitingForDrawer")  
        ]),  
    state(  
        "waitingForLight",  
        [],  
        [transition("lightOn","unlockedPanel")]),  
    state(  
        "waitingForDrawer",  
        [],  
        [transition("drawerOpened","unlockedPanel")]),  
    state(  
        "unlockedPanel",  
        ["unlockPanel","lockPanel"],  
        [transition("panelClosed","idle")])
)
Demo

• Concrete and abstract syntax

• Parse and implode
Transformation
Transformation in Rascal

- Functional programming
- Type-preserving visit
Visit

- Similar to switch
- Visits all nodes
- Specify nodes of interest only
- Structure shy
- Bottom-up, top-down, innermost, outermost
Print all state names

```java
visit (ast) {
    case state(str name, _, _): println(name);
}
```

do something when visiting a state
Suffix state names

```
ast2 = visit (ast) {
    case state(n, a, t) => state("<n>_", a, t)
        when size(n) % 2 == 0
}
```
Suffix state names

\[\text{ast2} = \text{visit} (\text{ast}) \begin{cases} \\
\text{case state}(n, a, t) \Rightarrow \text{state}(\"<n>_\", a, t) \\
\hspace{2em} \text{when size}(n) \% 2 == 0 \\
\end{cases} \]
Suffix state names

\[
\text{ast2} = \text{visit} \ (\text{ast}) \ {\text{case state}(n, a, t) \Rightarrow\text{state}("<n>_", a, t) \text{ when size}(n) \% 2 == 0}
\]

rewrite states

side condition
Suffix state names

updated tree

ast2 = visit (ast) {
    case state(n, a, t) => state("<n>_", a, t)
    when size(n) % 2 == 0
}

rewrite states

side condition

Wednesday, May 23, 12
Traversals strategies

default strategy

bottom-up visit (ast) {
    case state(str name, _, _): println(name);
}

top-down visit (ast) {
    case state(str name, _, _): println(name);
}
Constant propagation

innermost visit (exp) {
  case add(0, x) => x
  case add(x, 0) => x
  case sub(x, 0) => x
  case sub(0, x) => neg(x)
  case neg(neg(x)) => x
  case mul(0, x) => 0
  case mul(x, 0) => 0
  case mul(1, x) => x
  case mul(x, 1) => x
  case mul(min(x), min(y)) => mul(x, y)
  case mul(min(x), y) => min(mul(x, y))
  case mul(x, min(y)) => min(mul(x, y))
  case div(x, 0) : throw "div by zero";
  case div(0, x) => 0
  case div(x, 1) => x
}

repeat until stable
Exercise: desugaring

In particular, you should note that reset events aren't strictly necessary to express Miss Grant's controller. As an alternative, I could just add a transition to every state, triggered by doorOpened, leading to the idle state. The notion of a reset event is useful because it simplifies the diagram.

Exercise: desugar reset events. For each reset event to each state, add a transition to the initial state.
Warm up exercises

- Use visit to count number of states/ transitions
- Use visit to remove all transitions
- Use visit to rename all events in transitions
Warm up exercises

- Use visit to count number of states/ transitions
- Use visit to remove all transitions
- Use visit to rename all events in transitions

```java
public Controller desugarResetEvents(Controller ctl) {
    init = ctl.states[0].name;
    return visit (ctl) {
        case state(n, as, ts) => state(n, as, ts + 
            [
            transition(e, init) | e <- ctl.resets ])
    };
}
```